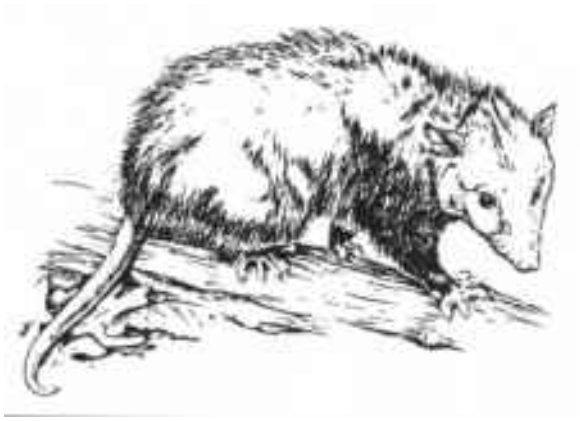
DARPA ITO/ATO:
FAULT TOLERANT NETWORKS

BBN TECHNOLOGIES
A Verizon Company

APOD Toolkit
Users' Guide



APOD release 3.0 - August 2002

http://apod.bbn.com

APOD Release History

August 2002	APOD release 3.0
April 2002	APOD release 2.1 - not cut
September 2001	APOD release 2.0
December 2000	APOD release 1.1
July 2000	APOD release 1.0
December 1999	APOD release 0.0

Document History

August 2002	Toolkit document 3.0
April 2002	Toolkit document 2.1
September 2001	Toolkit document 2.0

On the cover:
The Opossum. Opossums, as a group, are among the oldest, most primitive mammals of the New World. Some scientists call them "living fossils" because they have survived relatively unchanged for at least 50 million years. The brain case of the opossum is rather small in comparison to other mammals of a similar size and it is often reputed to be a rather dim-witted animal. While lacking in so-called "intelligence", the opossum is nevertheless an interesting and very successful animal. In response to a threat, an opossum is more likely to resort to a different tactic than biting: they pretend to be dead. This is actually quite comical to see: the animal will open its mouth and loll out its tongue, looking for all the world like it is actually dead. Since many predators will only eat live prey, this tactic of "playing possum" must be moderately successful.
See: http://www.opossum.org/

Copyright © 2002 by BBNT Solutions LLC.

Produced by BBN Technologies, A Part of Verizon.
10 Moulton Street, Cambridge, MA 02138, (617) 873-2000.

C O N T E N T S

Preface v

Purpose of this Document v

Background of the Project v

Technical Questions v

1 Introduction 7

2 Installing APOD 9

2.1 Components of the APOD Toolkit 9

2.1.1 Components Explained 9

2.2 Software Requirement for Using APOD 9

2.3 Installing APOD Binaries for Linux 10

2.4 Working with APOD Sources 10

2.5 Configuring your shell environment to work with APOD 11

2.6 Configuring your Linux system to work with APOD 11

2.6.1 Allow nopassword ssh access between hosts 12

2.6.2 Allow X11 remote access 13

2.6.3 Add hosts to /etc/hosts 13

3 Using APOD Defense Mechanisms 14

3.1 Linux firewall: "iptables" 15

3.1.1 Description 15

3.1.2 Requirements 16

3.1.3 Detailed Explanation 16

3.2 Lightweight Network based IDS: "snort " 18

3.2.1 Description 18

3.2.2 Requirements 18
3.2.3 Detailed Explanation 19
3.3 File based IDS: "Tripwire" 20
3.3.1 Description 20
3.3.2 Requirements 21
3.3.3 Detailed Explanation 21
3.4 Linux IPsec: "FreeS/WAN" 22
3.4.1 Description 22
3.4.2 Requirements 22
3.4.3 Detailed Explanation 22
3.5 Secured nameserver: "objectserver" 25
3.5.1 Description 25
3.5.2 Requirements 25
3.5.3 Detailed Explanation 26
3.6 Replication Management: "bus" 27
3.6.1 Description 27
3.6.2 Requirements 27
3.6.3 Detailed Explanation 27
3.7 Bandwidth Broker 29
3.7.1 Description 29
3.7.2 Requirements 29
3.7.3 Detailed Explanation 30
3.8 Port Hopping 31
3.8.1 Description 31
3.8.2 Requirements 31
3.8.3 Detailed Explanation 32
3.9 TCP Connection Flood Protection 33
3.9.1 Description 33
3.9.2 Requirements 33

3.9.3 Detailed Explanation 33
3.10 ARP Spoof Detection 34
3.10.1 Description 34
3.10.2 Requirements 35
3.10.3 Detailed Explanation 35
4 Prepackaged Reusable Defense Policies as Qoskets 36
4.1 Containment Qosket - breaking the flow of attacks 36
4.1.1 Mechanisms Used 37
4.1.2 Adaptation Policy 38
4.1.3 Customization Issues 39
4.2 Dynamic Reconfiguration / Outrunning attacks 40
4.2.1 Mechanisms Used 40
4.2.2 Adaptation Policy 40
4.2.3 Customization Issues 40
4.3 LAN Containment Qosket - controlling floods and block closer to the source 41
4.3.1 Mechanisms Used 41
4.3.2 Adaptation Policy 42
4.3.3 Customization Issues 43
5 APOD Example Applications 44
5.1 brokered_imgsvc 44
5.2 sniff_abatement 45
5.2.1 Stepping through the defense enabling process 45
5.2.2 Description 46
5.3 bette-3stage 47
5.3.1 Stepping through the defense-enabling process 47
5.3.2 Description 48
5.4 warfare_scenario2 49
5.4.1 Stepping through the defense-enabling process 49

5.4.2 Description 50
6 Future Work 51
6.1 Additional mechanisms 51
6.2 Additional Examples 51
6.3 Packaging defense-strategies into Qoskets 52
6.3.1 Change Server Object 52
6.3.2 Change of Network Routes 53
7 APOD People and Acknowledgement 54

Preface

Purpose of this Document

This manual documents the components and usage of the APOD toolkit. The APOD toolkit, developed under this project, provides a set of mechanisms, known as *defense mechanisms,* that can be used to *defense-enable* a distributed application. This document, together with the examples contained in the software release, is meant to describe how the defense mechanisms in the toolkit can be used to defense-enable an application,

This manual is part of a larger (and evolving) set of documentation about various technologies centered around the QuO middleware[1]. Therefore, you will frequently find references to the QuO *Users' Guide* and *Reference Guide*, which are available online.

Background of the Project

APOD is sponsored by DARPA ITO under the Fault Tolerant Networks program (which is currently under DARPA ATO), which seeks to develop focused technologies that support continued operation of network-centric applications in the presence of successful attacks.

BBN Technologies' APOD (Applications That Participate in their Own Defense) is providing a QuO-based toolkit for defense-enabling applications.

Technical Questions

Please direct any technical questions to quo-users@bbn.com.

[1] QuO is an advanced middleware framework for supporting Quality of Service based adaptation in distributed object computation (DOC), developed at BBN. Refer to http://quo.bbn.com

This page is blank intentionally.

1 Introduction

This project has developed *defense-enabling* technology by which a distributed application can be made more resilient to malicious attack. An intruder can gain access to and attack the computer systems on which the application runs, corrupting or disabling them. The technology developed in this project will allow the application to adapt to survive, and sometimes counter, these attacks. A defense-enabled application incorporates a *defense strategy* that it follows in defending itself against the attacks. The strategy is developed separately from the application's functional parts and usually involves the services of several mechanisms that are not directly related to the functional aspects of the application. These mechanisms are called *defense mechanisms.* The main contributions of this project are

1. Development of the concept of defense-enabling

2. Development of several defense mechanisms

3. Development of middleware support and an architectural framework to incorporate a defense strategy into an application

4. Examples showing how to use the defense mechanisms in defense enabling

5. Validation of defense strategies by experimentation, using an independent Red Team.

This document describes the defense mechanisms and how to use them by means of the example defense-enabled applications.

We use the QuO adaptive middleware as a basis for organizing the application's defense. The middleware allows an application to use a variety of mechanisms for sensing and responding to changes in environmental conditions. An application using QuO can sometimes sense malicious intrusion both by interacting with intrusion detection systems (IDSs) and by observing anomalies in its own behavior and in the environment.

Responses taken by an intrusion-aware application will range from changing the application's own behavior to adjusting resource requirements and attempt to influence the mechanisms that manage the resources. We claim that by adapting to, and trying to control, its environment an application can increase its chance of survival under attack.

Our approach to defense differs from, but complements, traditional security engineering. Traditionally, a secure system is built around an uncircumventable Trusted Computing Base (TCB) that enforces security for all applications in that system. Most operating systems and networks in use today, however, offer only imperfect protection that can be circumvented by intruders. Because of increasing complexity, commercial priorities, heterogeneity, and distributed nature of computer systems, this infrastructure is likely to remain vulnerable in the near future. On the other hand, we assume that existing security measures in that infrastructure -- limited access to privileged commands, IDSs, etc. -- serve to slow down attacks and offer warning of some of them. Our application-level and middleware defenses are intended to augment, not replace, these infrastructure defenses.

The defensive strategies we have developed use various services, of which the following are available with QuO [refer to the QuO documentations for details on QuO Services]:

- OODTE access control and cryptography-based security services to protect against direct corruption of the application;
- AQuA replication management service to counter attacks that destroy application component processes;
- The DIRM RSVP-based bandwidth management service to counter flooding attacks on network segments.

In addition, APOD has incorporated other security-related tools including:

- Linux IPTables, to filter network traffic at the individual hosts;
- The Tripwire and Snort intrusion detection systems (IDSs), to serve as complementary triggers for heightened alertness;
- The Free Swan security-enhanced IP, to harden the infrastructure on which applications run;
- A name- and object-server to provide security-enhanced lookup services to other mechanisms and to the application;
- A self-stabilizing software bus, to provide the distributed coordination needed by most defense strategies and to offer a light-weight alternative to the AQuA technology in QuO;
- A security-enhanced RSVP, to serve as a hardened replacement for the DIRM technology in QuO.

The APOD Toolkit packages the defense mechanisms in a modular and reusable manner, described in this document. Section 2 tells how to install the Toolkit. Section 3 tells how to use the defense mechanisms individually. Section 4 describes basic units of defense strategy, called qoskets, that combine the use of several basic mechanisms. Section describes several defense-enabled applications, the first of which was attacked as part of Red Team validation experiments. Finally, section 6 mentions a few ways in which the APOD Toolkit could be enhanced.

2 Installing APOD

2.1 Components of the APOD Toolkit

The APOD release consists of the components indicated by checkmarks in the table below:

Component	Linux i386 Tarball	Source Tarball
ApodDefense	☑	☑
ApodDoc	☑	☑

Since APOD is build on top of the QuO middleware, the following components are required for APOD

- quoCore
- quoOODTE
- quoRsvporb

The QuO components are available from http://quo.bbn.com

2.1.1 Components Explained

ApodDefense	Defense Mechanisms, including iptables, snort, bus plus examples demonstrating their use
ApodDoc	APOD User's Manual and generated documentation

2.2 Software Requirement for Using APOD

The APOD software has been successfully tested on the platform:

- Complete QuO release >= 3.0.10

- RedHat 6.2
- gcc 2.95.2
- Bourne or CSH compatible shell
- GNU make, version > 3.78.1
- JacORB, version 1.3.30
- JDK 1.3.0
- X Windows for GUI based applications
- openssh
- sudo
- quoCore (http://quo.bbn.com)
- quoOODTE (http://quo.bbn.com)
- quoRsvporb (http://quo.bbn.com)

In addition, each mechanism has its own set of dependencies to 3rd party software. See section 3 for further details.

2.3 Installing APOD Binaries for Linux

First make sure you've installed all necessary components from the QuO release, ie quoCore and quoOODTE. See the QuO User's Guide section 2 for further details.

You can install the APOD software from a set of tar balls in a custom location of your choice:

```
> tar xvfz  apod*.tgz
```

2.4 Working with APOD Sources

APOD sources are released as a set of tarballs. After downloading the tarballs, unpack them by

```
> tar xfvz apod*.src.tgz
```

The APOD Makefiles follow already existing coding standards as close as possible. For JAVA, easy to understand makefiles using standard utilities (javac, GNU utilities) were written.

For using APOD in your own application, you are free to choose whatever makefile paradigm you prefer. However, we recommend modeling the makefiles for QuO specific parts (*i.e.*, QDL code generation) after the already existing makefiles contained in the release. For a detailed description of the makefile structure for $QUO_ROOT/examples/simple, see the *Quo Toolkit Reference Guide*.

2.5 Configuring your shell environment to work with APOD

You have to source the following scripts, which setup your environment:

To setup QuO:

```
> cd ~/quo/scripts/environment
> source quoenv.sh ~/quo
```

To setup APOD:

```
> cd ~/quo/apod/scripts/environment
> source apodenv.sh ~/quo/apod
```

Note: similar scripts exist for the users of csh based shells. In general, it's a good idea to put these scripts into your shell's initial startup scripts (like .bashrc or .cshrc).

2.6 Configuring your Linux system to work with APOD

After building APOD as described above, depending on how your host system and environment is configured, there may a couple of pitfalls one need to overcome before you can get APOD software running in an out-of-the-box way. Note that APOD does NOT require you to have this configuration, but it is our experience that the following makes it easier.

There is a script that will install most of the third party software you will need to run the APOD examples. The software program sudo is installed and configured to give the APOD mechanisms the ability to run the command that require root privileges.

The 3rd party software list includes:

- JacORB 1.4.1
- J2SDK1.4.1_rc1
- Secure RSVP(SRSVP) 2.2
- Tripwire 2.2.1
- Iptables 1.2.5-3
- SRSVP-CMS
- Openssh 3.1p1-6
- Snort 1.8.1-2
- SRSVP-PolicyServer

- NaiOODTE 1.2-1
- Sudo 1.6.5p2-2
- Iproute 2.4.7-1

To run this installation script, follow the instructions below.

Get the APOD third party tarball.

user > tar -xfvz RH73_APOD_3rdparty.tgz

As root:

```
root > cd $APOD_ROOT/scripts/apodbuild

root > ./install_apod \

        --3rdPartyDir <path to RH73_APOD_3rdparty/> \

        --user <userlogin> \

        --envdir </tmp> \

        --installall </opt>
```

userlogin – is the user who will run APOD and the environment script will be make for

envdir – is where to place the environment scripts when they are created

installall – can be used to specify a root to place some 3rd party software

checkonly – won't actually install anything, just check it

Once the script is done, the environment scripts generated should be sourced to set up the needed environment variables for running APOD.

user> source envdir/apodenv.sh <apod installation directory (eg. ~/quo/apod)>

2.6.1 Allow nopassword ssh access between hosts

APOD scripts for starting up distributed applications rely on the fact that once logged into a system as a regular user, that user can ssh into any other APOD enabled host without being asked for a password. The standard way to achieve this is to generate a keypair via

```
> ssh-keygen
```

Enter an empty passphrase.

Next do

```
> ssh-keygen -d
```

Again, enter an empty passphrase.

Next, do

```
> cat ~/.ssh/identity.pub >> ~/.ssh/authorized_keys
> cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys2
```

Next, tar up the whole .ssh directory via

```
> tar cfvz ~/ssh.tgz ~/.ssh
```

transfer this tarball over to all the other hosts, and untar it in the user's homedirectory via

```
> tar xfvz ~/ssh.tgz
```

Try to ssh between the hosts, and you should not be asked for a password anymore (after initially asking you permission to add in the remote host into an internal list).

2.6.2 Allow X11 remote access

This is easy to achieve: Just make sure you have a line like

```
> xhost +host1
> xhost +host2
> ...
```

in your shells init scripts (.bashrc / .cshrc).

Test this by doing something like

```
user@host1> xterm -display host2:0.0
```

which should pop up an xterm on host2's display.

2.6.3 Add hosts to /etc/hosts

APOD scripts rely on having all machines listed in /etc/hosts if no DNS is used.

3 Using APOD Defense Mechanisms

The APOD release contains a set of *defense mechanisms*. A *defense mechanism* offers a service or a capability that is used in the *defense strategy* of a *defense-enabled* application. Any mechanism that offers a feature that is relevant for the security or survivability of an application can be used as a defense mechanism in defense-enabling that application. By means of the example defense mechanisms we hope to demonstrate how to one can take an external mechanism and use it as a defense mechanism in the context of defense-enabling.

The defense strategy of a defense-enabled application is coordinated by the QuO adaptive middleware. This coordination requires that the mechanism has to interface with QuO by means of qoskets that might contain system conditions, callbacks, and contracts. We assume familiarity with the concepts and constructs of the QuO adaptive middleware.

In many cases, the core mechanism itself already exists as COTS software. The APOD release does *not* contain those third party components, but provides wrappers to utilize them as defense mechanisms.

The following sections describe the mechanisms that are included in the APOD release. These are the defense mechanisms that are specifically developed for APOD; they may or may not wrap a COTS tool or mechanism. Apart from these, other mechanisms that we have used with the QuO middleware such as bandwidth management using DIRM, or replication management using AQUA, or access control using OODTE can also be used as defense mechanisms.

3.1 Linux firewall: "iptables"

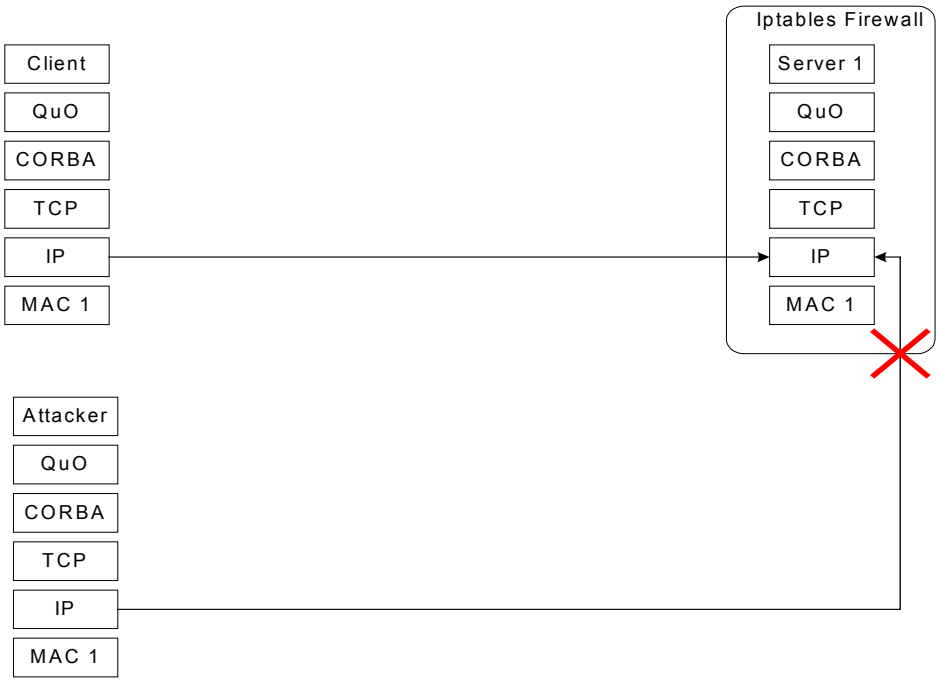
3.1.1 Description

The "iptables" mechanism provides an interface to the standard Linux firewall system. It is an actuator that allows a defense-enabled application to

- Make use of the full flexibility of iptables by dynamically executing any command
- Use prepackaged dynamic behavior, such as dropping packets from a source, blocking new connections from source, limit the rate of incoming TCP connections, reject all new outgoing connections from a specific user.
- Use prepackaged static firewall configurations.

This firewall-based mechanism operates on the IP layer, as shown in the following figure:

Figure 1 Scheme of the packet filtering mechanism



3.1.2 Requirements

1. Operating System: Linux 2.4.x with iptables installed

2. System configuration: sudo access for "iptables" setup with nopasswd option.

3.1.3 Detailed Explanation

The mechanism contains the following parts (see the corresponding IDL files for a more complete interface description):

- IptablesManager: CORBA wrapper around the Unix command "iptables"

- IptablesValueSC: QuO system condition that allows people to make raw calls on the IptablesManager from QuO

- PresetFirewallSC: QuO system condition that encapsulates predefined rules set, which can be activated through the method set_firewallconfig according to restrictiveness.

- DynamicAttackResponseSC: QuO system condition that allows dynamic changes to the rules sets. This syscond also has a builtin whitelist capability to prevent certain hosts from ever being blocked.

Here a description of the methods of the DynamicAttackResponseSC:

- `ipdrop (in string sourceip, in state onoff)`
  used to drop IP packets from a source

- `ipdrop_time (in string sourceip, in long long duration)`
  version of ipdrop that drops for duration seconds

- `ipblock (in string sourceip, in state onoff)`
  a more polite IP blocker

- `ipblock_time (in string sourceip, in long long duration)`
  version of ipblock that blocks for duration seconds

- `tcp_limit (in long port, in long rate, in timescale scale, in state onoff)`
  limits rate of incoming TCP connections

- `user_outblock (in long UID, in state onoff)`
  rejects all new outgoing connections by user

- `addToWhiteList (in string host)`
  prevent host from ever being blocked

- `removeFromWhitelist ( in string host)`
  allow host to be blocked again

After installing the mechanism, you can defense-enable your application by using the sysconds provided in this mechanism. This typically involves generating a QuO contract and/or QuO Delegate that makes calls on the sysconds. The sysconds, in turn, make calls on the manager component, which executes the Linux "iptables" command. Note that the sysconds and the manger are not necessarily collocated on the same machine. It is necessary to start the manager on the host on which you want to control the firewall, via

```
> $APOD_ROOT/mechanisms/bin/IptablesManagerImpl -naming.class
<nclass> -naming.properties <pfile>
```

with the <nclass> being a supported naming class and <pfile> being the property file used for naming purposes.(see documentation of objectserver).

The following figure shows the interactions between the two components syscond and manager. The syscond is in the context of a contract that is attached to an object (marked "Client") that makes CORBA call to another CORBA object (marked "Object").

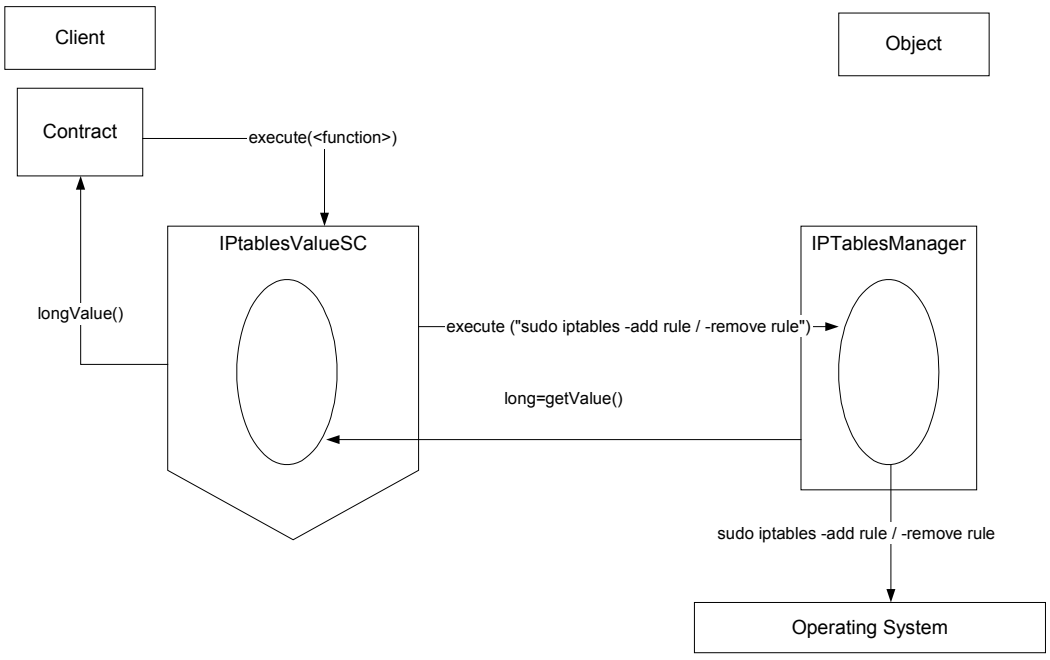


Figure 2 Organization of the IPTables based packet filtering mechanism

3.2 Lightweight Network based IDS: "snort "

3.2.1 Description

The "snort" mechanism is a "sensor" that interfaces with the lightweight network based intrusion detection system. It makes the services of a network based intrusion detection system available for use in defense-enabling. Using it a defense-enabled application can be aware of:

- general signature based attacks on a particular host or a set of hosts on a network segment
- application specific anomalies concerning specific hosts, ports or TCP/IP protocols (such as an ICMP packet on a port on which a CORBA server is listening)

This awareness can then be used to trigger defensive response.

3.2.2 Requirements

1. Snort 1.8.1 from http://www.snort.org
2. System configuration: sudo access for "snort", see snort mechanism's README for more information.

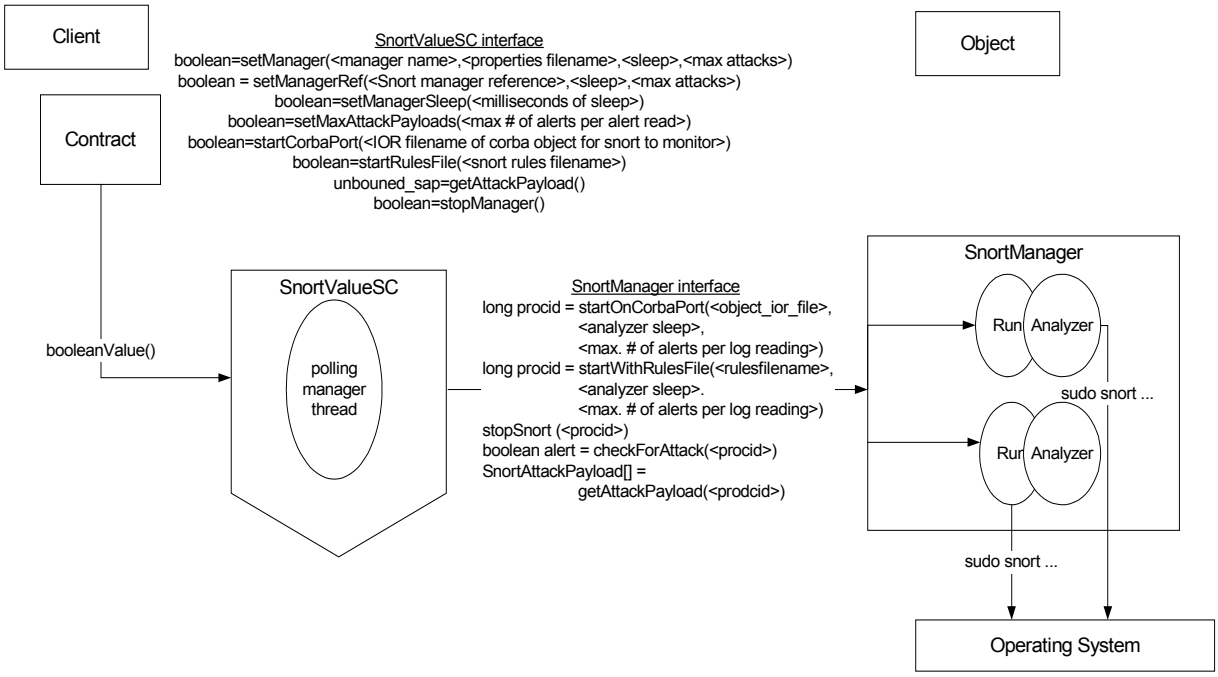


Figure 3 Organization of the lightweight network based IDS mechanism

3.2.3 Detailed Explanation

The mechanism contains the following parts (see the corresponding IDL files for a more complete interface description):

- `SnortManager: CORBA wrapper around the Unix command "sudo snort …"`
- `SnortValueSC: QuO system condition that allows people to make calls on the SnortManager from QuO`
- `SnortLogAnalyzer: The SnortManager will start an instance of this threaded class for each Unix command "sudo snort"`

Figure 3 describes the interaction between the syscond and the manager in the context of a contract, which is associated with a CORBA client.

APOD's snort mechanism comes with a set of snort rules. These rules are from the snort website and have been modified to record alerts in a readable format for the log analyzer. There is documentation on the snort website on writing rules. We suggest writing new rules into a new rules file and adding an include statement in the rules/apod.rules file. There is an APOD alert type defined in the rules/apod.rule file. Any new rules will need to use this apod alert type or the log analyzer will not recognize the logged output from the rules.

3.3 File based IDS: "Tripwire"

3.3.1 Description

The “Tripwire” mechanism is a “sensor” that interfaces with the Tripwire intrusion detection system, which is a file-system integrity checker. Using this sensor, a defense-enabled application can be aware of:

- Changes in the permission and mode bits or in the type or size of files
- Changes in the file content
- Addition or deletion of files and directories

The idea here is that some attacks in some stage of their progress have these kinds of effects. By being aware of these effects, a defense-enabled application can take some action in response. As an example, one can use tripwire mechanism to monitor the file system segment that stores the application’s executables and in response to an indication that the some file there has been modified, the application may decide not to start any new application component on that host to prevent starting a component with compromised executable.

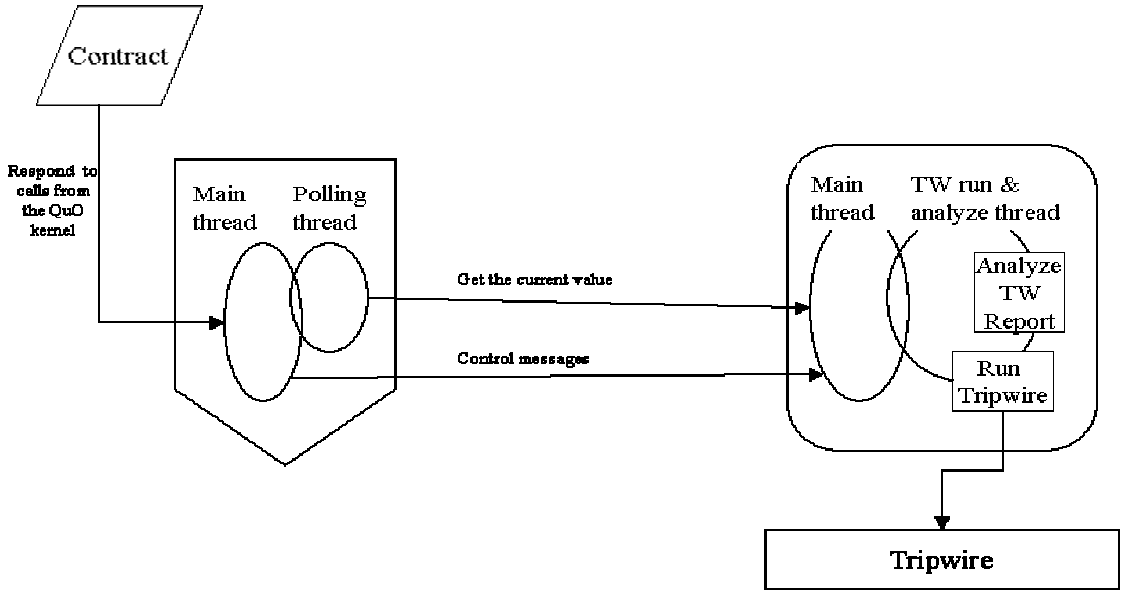


Figure 4 Organization of the file based IDS using Tripwire

3.3.2 Requirements

1. Operating system: Linux 2.4.x

2. Tripwire for Unix version 2.2.1: you need to know the install directory, location of the site key file and the local key(s) for the host(s) on which you will run Tripwire.

3.3.3 Detailed Explanation

The tripwire mechanism consists of a Tripwire manager that controls and interacts with the Tripwire system, and a system condition that bridges the manager with the QuO framework. The organization is shown in Figure 4.

The Tripwire manager provides an interface for initiating single or periodic scanning by Tripwire, switching to a different policy file (which will be used whenever the next scan happens) etc. Each host with a physical file system can have a Tripwire manager.

In order to use the Tripwire mechanism in the middleware-based adaptation, a middleware component, namely a system condition, needs to interact with the Tripwire manager. Currently we have a system condition implementation where it is hooked up with one single Tripwire manager. This is appropriate for a distributed setting where the adaptation control and support is decentralized as opposed to having all Tripwire managers hooked up with a single system condition.

For further details about the interfaces of the system condition refer to the corresponding IDL files in the tripwire subdirectory in the mechanisms directory of the release.

3.4 Linux IPsec: "FreeS/WAN"

3.4.1 Description

This is an “actuator” that interfaces with the low-level IP communication mechanisms so that transparent switching between IP and IPSec is possible as part of a defense-strategy. This mechanism uses FreeS/WAN (www.freeswan.org), for its IPSec communications.

Figure 5 Organization of the mechanism using Linux IPSec

3.4.2 Requirements

1. Operating System: Linux with FreeS/WAN 1.90 installed and with a tunnel configured. FreeS/WAN can be found at www.freeswan.org/download.html#release Instructions on installing FreeS/WAN are found within the download and will require a kernel recompile as they add features to the kernel.

2. System configuration: sudo access for "ipsec"

3.4.3 Detailed Explanation

The mechanism contains the following parts (see the corresponding IDL files for a more complete interface description):

- `IPsecManager: CORBA wrapper around the Unix command "ipsec"`

- `IPSecValueSC: QuO system condition that allows people to make raw calls on IPsec from QuO`

After installing the mechanism, you can defense enable you application by using the sysconds provided in this mechanism. This typically involves generating a QuO contract and/or QuO Delegate that makes calls on the sysconds. The sysconds, in turn, make calls on the manager component, which executes the linux "ipsec" command. Note that the syscond and the manger must be on opposite ends of the IPsec tunnel. You can start the ipsec manager as shown below.

```
> $APOD_ROOT/mechanisms/bin/IpsecManagerImpl -propfile <prpfile>
```

with the <prpfile> containing a tag of ObjectServerURL to find the objectserver's IOR, and a tag of profile that specifies a profile to use with FreeS/WAN.

There are seven methods in the IPSec manager interface that are used

to control the underlying FreeS/WAN mechanisms. These are called using sudo so you will need to be able to execute /usr/local/sbin/ipsec for this mechanism to be useful. FreeS/WAN 1.91 and 1.90 behave differently and will require different methods to be called to lower a connection across a network. FreeS/WAN 1.91 won't let you unroute if a connection is currently active while 1.90 will.

There are seven methods provided by this mechanism. Five of them are atomic ipsec commands while two are combinations of two calls and are used to lower connections across a network and depend upon the version of FreeS/WAN being used. These methods are:

- up(profile) will raise the tunnel specified by the profile name that is passed in. This profile needs to be specified in /etc/ipsec.conf. This "turns on" IPsec and encrypts communications between the endpoints specified by the profile.

- down(profile) will tear down the specified tunnel. This turns off the encryption of messages using IPsec between the endpoints. Be aware that messages will not start to flow in the clear after a call to down until an unroute(1.90) or delete(1.91) command has been sent. This is to prevent messages from passing in the clear accidentally. To allow unencrypted messages to pass unroute or delete needs to be called after down.

- unroute(profile) is used with FreeS/WAN 1.90 and will restore the connectivity between two hosts that have had their tunnel taken down. The traffic will no longer be routed through ipsec (which quietly drops the messages when the tunnel is down but has not yet been unrouted) and will pass as it did before the tunnel was brought up.

- replace(profile) is used with FreeS/WAN 1.91 and does a similar job to unroute. It removed the given profile from the IPsec process and adds it again. When it is added it's in an unencrypted mode and therefore replaces routing that a down will not.

- delete(profile) will remove a specification from the IPsec process and unroute and connections that it may have been using.

- downAndUnroute(profile) is used with FreeS/WAN 1.90 and calls down and then unroute on a connection in a single call. This is needed if the caller is across a tunnel from the machine that will lower the tunnel. If the called tried to make two separate calls, one to down and the next to unroute the tunnel, the unroute call would never get through as the route would still point to the tunnel that no longer was working.

- downAndReplace(profile) is similar to downAndUnroute but works with FreeS/WAN 1.91. It calls down and then unroute, thereby restoring unencrypted connectivity. Replace is a stronger unroute that will work with FreeS/WAN 1.91 and unroute a tunnel even if a connection is active.

The IPSecValueSC is a rather straightforward mapping from the functionality of the IPsec manager. It has 9 methods. Four of these methods map directly to the manager: These functions take no arguments on account of the fifth method, setTunnel(string).

- `raiseTunnel()` brings a tunnel up

- `lowerTunnel()` brings a tunnel down

- `untouteTunnel()` brings routing back to normal

- `deleteTunnel ()` deletes a references to a tunnel from IPsec

- `replaceTunnel ()` puts the tunnel definition back to its original state

- `lowerAndUnrouteTunnel()` does both lower and unroute when doing only one would render the system unable to do the other. This is used with FreeS/WAN 1.90

- `lowerAndReplaceTunnel ()` does both lower and replace when doing only one would render the system unable to use the other. This is used with FreeS/WAN 1.91

- `setTunnel (in string tunnel)` this method saves the name of the tunnel and uses it when a tunnel name is needed to be passed to the manager. This lets you only set the name once and make many calls.

- `setManager (in string objectServerURL)` is used to tell the manager the IOR of the objectServer. It takes in a URL that points to an IOR of an object server. This allows the system condition to get a reference to a manager.

3.5 Secured nameserver: "objectserver"

3.5.1 Description

This mechanism provides service analogous to directory service (such as CORBA naming service), but in a secured manner and is intended for serving up object-references, public keys and even signed class files. Only the serving the CORBA interoperable objects references are currently implemented. We needed this auxiliary service because we needed a more secured way to obtain object references than the standard CORBA way (IOR files or un-authenticated naming service).

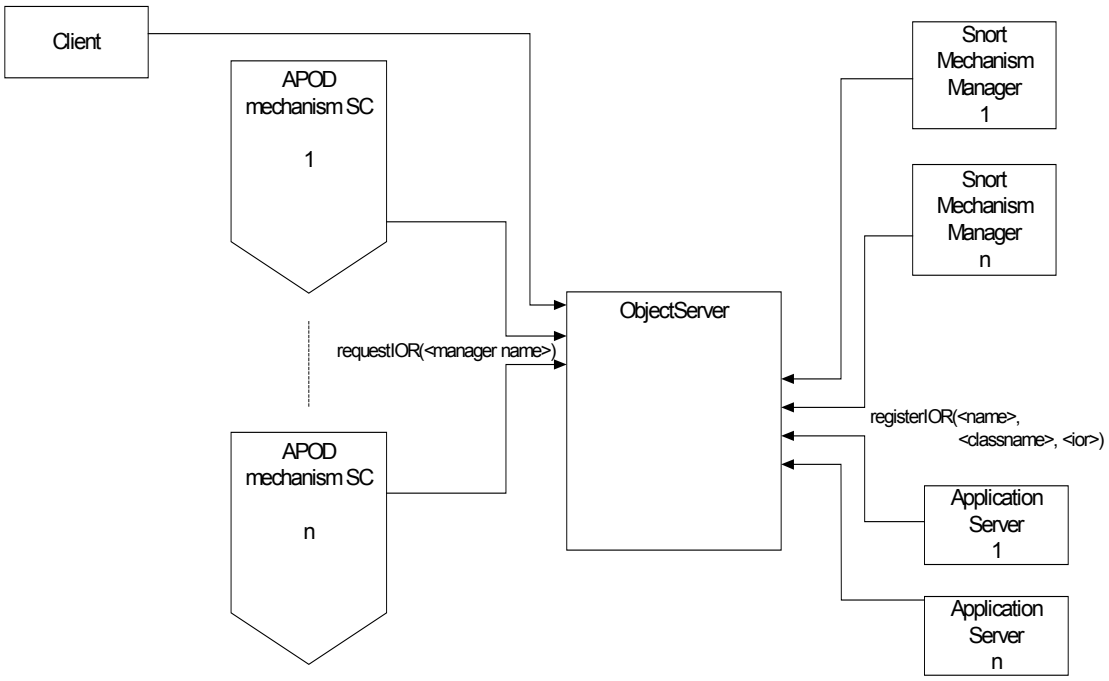


Figure 6 Schematic diagram of the Object Server

3.5.2 Requirements

1. Availability of an HTTP server to serve the reference of the object server
2. OODTE

3.5.3 Detailed Explanation

A schematic diagram of how we use the object server is shown in Figure 6. Object server itself is a CORBA objects that starts up and writes its IOR in a place that can be served as an URL. This allows other components to bootstrap with the object server. This object exposes an interface to register object references, keys etc and retrieving them. OODTE is used for access control: one can register or retrieve only if it has the right cryptographic credential. For further details about the interface of object server, refer to the IDL definition in the objectserver subdirectory of the mechanisms directory in the release. Any component that wishes to use its service firsts uses HTTP to obtain a reference to the object server. It then uses this object like standard CORBA server.

Also in the ObjectServer mechanism is a supporting naming interface and three implementations. The naming interface defines binding and resolving methods. The naming interface allows the user to specify which naming implementation to use at startup time. The implementations we have already developed are File, FileRemove, and ObjectServer. The File naming bind method will write an object reference to a file and the resolve method will read that file. The FileRemove implementation is similar except the resolve will remove the file once read. The ObjectServer naming bind method will get an ObjectServer URL reference and register the given object. The resolve will also gain an ObjectServer URL reference and then request an object reference by name.

Using the APOD naming interface requires two command line arguments, a naming.class and naming.properties. The File and FileRemove implementations need a property named ‘Directory’ and its value is the directory where the object reference file is written and read. The ObjectServer implementation requires a property named ‘ObjectServerURL’ and its value is the URL of the ObjectServer. The ObjectServer README has more information on how to use APOD naming.

3.6 Replication Management: "bus"

3.6.1 Description

The self-stabilizing software bus, or "bus" for short, is an APOD mechanism for publishing data about the status of the APOD defenses and of the applications they defend. It can also be used to maintain a specified number of replicas of application processes.

The bus has the property that data published at one place will eventually be available everywhere. In this way, it is analogous to a hardware bus, but implemented in software. Any bus process, i.e., a process that has subscribed to the bus, can publish data that will eventually be seen by every other bus process.

The bus is also self-stabilizing in the sense defined by Dijkstra: an arbitrary change to the bus data will eventually lead to a stable state in which all functioning bus processes agree on the bus data.

Thus the bus is inherently fault-tolerant: some kinds of failure, including crashes of bus processes and corruption of its data, are corrected by the bus mechanism as it stabilizes.

One side effect of the bus mechanism is that, in a stable state, every functioning bus process knows the bus data of every other bus process. This fact makes it easy for the bus to manage process replicas: whenever there are too few replicas of one kind of process, a new replica is created. To support replication management, the bus publishes specifications that tell how many processes of each kind are to be maintained and constraints on where new processes can be created.

Like any other bus data, these specifications can be changed dynamically and changes will propagate everywhere on the bus.

3.6.2 Requirements

1. Operating system: Unix (tested only on Linux)

2. CORBA for Java (tested only with JacORB)

3.6.3 Detailed Explanation

To subscribe to the bus, a process must first initialize the bus with a call to the static method com.bbn.quo.bus.Bus.init, which takes two arguments:

- the ORB to use for bus communication;

- the name of a file containing bus configuration parameters.

If the name of the configuration file is null, or if a call is made instead to an alternate form of the init method without this name, then the name of the configuration file is taken from the Java property named "bus.properties" .

Optionally, the process may then publish the kind and the IOR of an application object it owns, using a call to the static method com.bbn.quo.bus.Bus.initApp. The arguments are:

- the kind of the application process (an arbitrary String);
- the CORBA Object whose IOR will be published along with other data about this process.

After initializing the bus, the process subscribes to the bus by instantiating the com.bbn.quo.bus.Proxy class. The constructor for this class takes no arguments, but some alternate forms for the constructor take arguments to supply to the initApp method, and these alternate forms make it unnecessary to call initApp separately.

A process can subscribe multiple times by constructing several Proxies. After the first subscription, the no-arg Proxy constructor must be used, because the bus initialization parameters cannot be changed once set.

A process should be able to unsubscribe from the bus by letting every Proxy object be finalized, but this feature does not currently work.

The bus configuration parameters, supplied in a disk file, include:

- the name of a file (default: "host.ini") containing an initial specification of host data, e.g., which hosts are unsuitable for application replicas, which hosts should be shut down, etc.;
- the name of a file (default: "kind.ini") containing an initial specification of process kinds, e.g., a list of kind names and a degree of replication for each, etc.;
- the name of a file (default: "/tmp/bus.ior") containing an IOR of another bus process that this process can contact to join the bus;
- the level of diagnostic detail to show, with zero being the least and increasing with larger integers.

Each of these parameters has a default value that is used if the parameter is not supplied or if the name of the configuration parameter file is null. Examples of the configuration parameter file and the files specifying hosts and process kinds can be found in the $QUO_ROOT/apod/mechanisms/bus/test directory.

The above explanation is enough to start any number of bus processes on a single host. The first process, when subscribing to the bus, will cause the /tmp/bus.ior file to be written to disk with its own IOR. New bus processes on that host will read /tmp/bus.ior to find the IOR of another bus process with which to share data. To start processes on another host, however, you must copy the /tmp/bus.ior file to that host by some non-bus means before starting any bus processes there. Once the /tmp/bus.ior file has been copied to a host, bus processes started on that host can join the bus, including sharing data with processes on other hosts. Once the /tmp/bus.ior file has been copied, the bus processes on its host will maintain it.

Once a process has subscribed to the bus, it uses the com.bbn.quo.bus.Proxy interface to read and modify the bus data.

The BusSCImpl class implements a QuO SysCond interface to the bus. This interface allows a subset of the bus data to be queried and modified remotely using CORBA invocations.

Because any bus process has access to all the bus data, any process can be used as a probe to monitor and control the bus. A simple user interface program, named "UI", is supplied for this purpose. "UI" offers a text interface for viewing all the bus data and for modifying a subset of that data.

3.7 Bandwidth Broker

3.7.1 Description

The Bandwidth Broker allows applications to securely request changes in the queuing policies of dedicated routers that affect how data flows through the routers among various network segments. The idea here is to manage the available capacity of the network paths to the application's advantage much like IntServ or DiffServ services could be used. It was implemented as a potential replacement for RSVP/SERSVP (which was exercising the filtering rules in the routers to establish reservations) based defense that we have experimented with earlier for the following reasons:

- Most RSVP implementations don't implement aggregation of multiple flows into a single filtering rule. The Bandwidth Broker allows this, which is helpful in supporting more complicated bandwidth based defense strategies where the requirement is based on aggregate flows between network segments.

- RSVP daemons are easy to crash. The Bandwidth Broker utilizes ssh for remote method calls.

3.7.2 Requirements

- Linux OS on client system

- Linux OS on routers

- ruby 1.6.6

- ssh access without password between client and bandwidth broker hosts, and bandwidth broker and routers

3.7.3 Detailed Explanation

The Bandwidth Broker is implemented with the following scripts:

setup_bb.rb:

This command is used to create the initial state of the bandwidth broker. Customize it to contain information about your specific network topology. Then simply run

```
bbroker_host> cd $APOD_ROOT/mechanisms/bbroker/
bbroker_host> setup_bb.rb
```

bbroker.rb <cmd> <source> <destination> <bw_rsvp> <bw_rest>

<cmd> = add | del | stat | clean
<source> = fully qualified IP address xxx.xxx.xxx.xxx/xx
<destination> = fully qualified IP address xxx.xxx.xxx.xxx/xx
<bw_rsvp> = bandwidth for reservation in bps, also understands xkbit xmbit
<bw_rest> = bandwidth for rest traffic in bps, also understands xkbit xmbit

bboker.rb implements the bandwidth broker logic. It is mainly responsible for dispatching the configuration calls to all routers and doing error handling in the presence of ssh timeouts.

Usage examples are:

```
bbroker_host> bbroker.rb add 10.0.0.103/32 10.0.0.32/28 600kbit 10kbit
bbroker_host> bbroker.rb stat
bbroker_host> bbroker.rb add 10.0.0.103/32 10.0.0.48/28 600kbit 10kbit
bbroker_host> bbroker.rb stat
bbroker_host> bbroker.rb del 10.0.0.103/32 10.0.0.48/28 600kbit 10kbit
bbroker_host> bbroker.rb stat
bbroker_host> bbroker.rb clean
bbroker_host> bbroker.rb stat
```

The above calls would usually be executed from via ssh such as:

```
host1> ssh bbroker_host "bbroker.rb stat"
```

It could be invoked from command line or from an application component such as a system condition.

tc_cmd.rb <cmd> <source> <destination> <bw_rsvp> <bw_rest>

<cmd> = add | del | stat | clean
<source> = fully qualified IP address xxx.xxx.xxx.xxx/xx
<destination> = fully qualified IP address xxx.xxx.xxx.xxx/xx
<bw_rsvp> = bandwidth for reservation in bps, also understands xkbit xmbit
<bw_rest> = bandwidth for rest traffic in bps, also understands xkbit xmbit

tc_cmd.rb is responsible for driving the "tc" command to configure Linux routers. It also has some logic to parse netstat -nr output to figure out which ethernet device to configure according to the routing table. This command will be invoked by the bbroker.rb over ssh.

Usage examples are:

```
router> tc_cmd.rb add 10.0.0.103/32 10.0.0.32/28 600kbit 10kbit
router> tc_cmd.rb stat
router> tc_cmd.rb clean
router> tc_cmd.rb stat
```

3.8 Port Hopping

3.8.1 Description

When a client communicates with a server over TCP/IP, all packets exchanged between the two parties contain a source <address:port> and target <address:port> pair. This is sensitive information, and attackers spent time sniffing network traffic in order to get addresses of potential targets. Traditional security mechanisms try to thwart attackers by encrypting and encapsulating data packets using VPN technologies like IPsec. However, after encrypting the payload information and for instance encapsulating the packet, the encrypted data is still sent between two dedicated endpoints. This leaves attackers the possibility of detecting endpoints via traffic analysis and running attacks against the endpoints.

Port and Address Hopping is an attempt to obfuscate the "real" ports and addresses by replacing them with values picked randomly from a range of possible values. Packets intercepted by attackers will only reveal the random addresses, which are valid only for a limited time period, i.e. 5 minutes. In order for attacks against the server to be successful, the attacker has to break into the refresh cycle and complete reconnaissance and execution within the cycle time. Furthermore, attempts to use ports that are different from the one picked randomly by the hopping mechanism, or where used in previous cycles raise red flags.

3.8.2 Requirements

NAT gateway implementation:

- NAT gateway running Linux 2.4.x with iptables and ruby
- NT / Linux clients and servers

User Tunnels:

- NT / Linux clients and servers

3.8.3 Detailed Explanation

The hopping mechanism is implemented by a client component and a gateway. The client component is located either on the client itself. It intercepts all communication to the real server, and replaces its <realaddress:realport> pair by <fakeaddress:fakeport>. The gateway component is located either on the server's LAN or the server itself. It does the reverse mapping from <fakeaddress:fakeport> to <realaddress:realport>. The <fakeaddress:fakeport> pair is picked randomly from a range of IP addresses and ports.

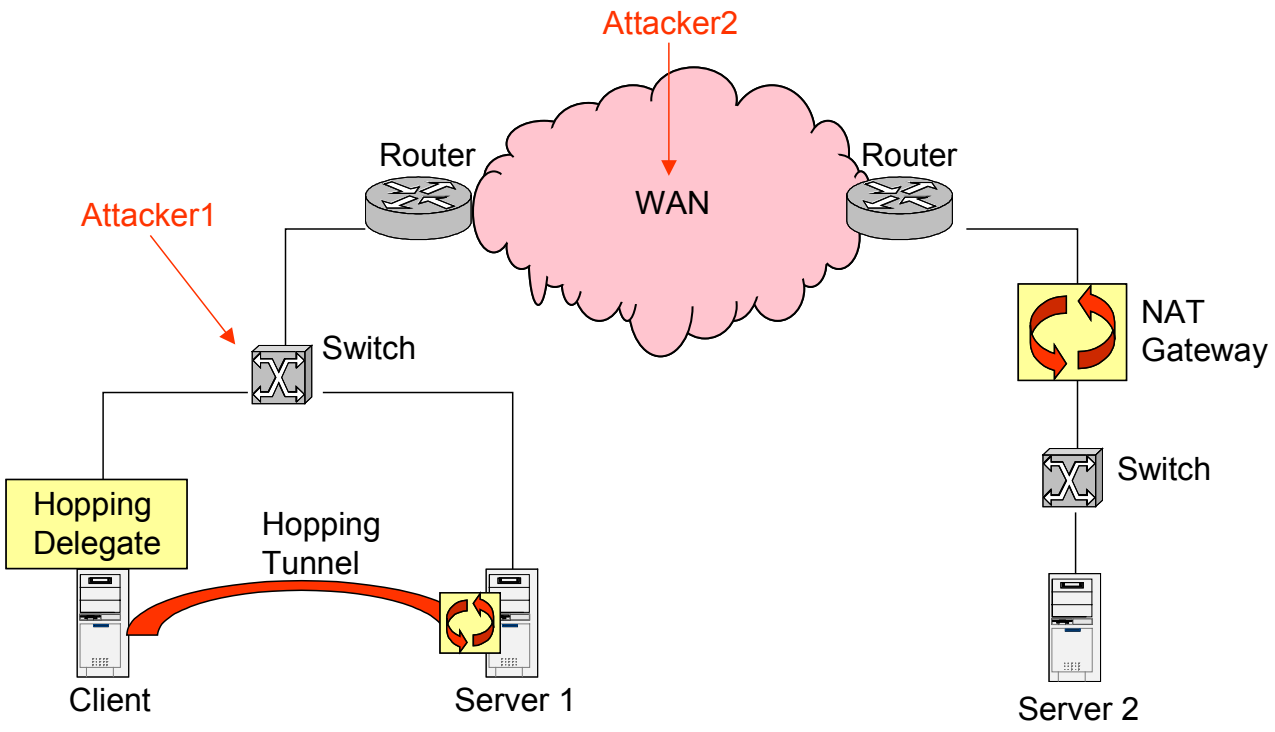


Figure 7 - Address / Port Hopping via Tunnels and/or NAT Gateways

The pair is used for a specific cycle time, after which a new pair is generated and used. Information about the previous pair is saved off, in order to be able to identify suspicious traffic using historic information.

Note that this mechanism relies on synchronization of the random number generators used in the two components, which can be achieved by seeding both generators with the same value. In addition, time synchronization is required to synchronize the switchover to using a newly generated <fakeaddress:fakeport> pair.

The implementation can be found in $APOD_ROOT/mechanisms/porthopping.

See the README for a more detailed description on how to use the ruby scripts.

3.9 TCP Connection Flood Protection

3.9.1 Description

Connection Flood Detection: the results from the first APOD redteam experiment showed that a connection flood to a port of a CORBA application would break the communication with valid clients and servers. In response to this attack, we developed a mechanism to counter these connection floods. The mechanism keeps count of the number of connections to specified application ports and raises an alert when a connection count exceeds a specified threshold. The mechanism has been implemented and tested in the containment qosket. The strategy implemented in the bus_containment qosket takes the offending remote host that has too many connections to a port and does a local block of all ip traffic to and from that host.

3.9.2 Requirements

1. Operating system: Unix (tested only on Linux)

2. CORBA for Java (tested only with JacORB)

3.9.3 Detailed Explanation

The TCP connection flooding defense is implemented in the ConnectionFloodSC syscond. The idl definition ($APOD_ROOT/qosket/bus_containment/idl/ConnectionFloodSC.idl) defines the following methods:

interface ConnectionFloodSC:

- syscond to act as a consumer for source ip address of violating packets
  void setSourceSC (in quo::ValueSC snortsource);

- neglect offenses from these IP addresses
  void setWhitelist(in whitelistarray whitelist);

- allows setting of vital parameters through use of property files
  void init (in java::util::Properties prop);

- how many seconds to sleep between each measurments
  void setAgility (in long sec);

- raise alert if the number of open connections to a port exceeds maxEstablished
  void setConnectionThreshold (in long maxEstablished);

- add the port of the IOR argument to the list of monitored ports
  void addObserveObject (in string ior);

- return a list of offenders
  quo::strings getOffendingHosts ();

- not in use currently, but allow direct coupling with iptables
  void hookupActuator (in apod::mechanism::iptables::DynamicAttackResponseSC iptablesSC);

The typical use is within a QuO contract. Check $APOD_ROOT/qosket/bus_containment/qdl/BusContainment.cdl for an example of how to hook up the connection flood detection with iptables to block offending sources.

3.10 ARP Spoof Detection

3.10.1 Description

Arp Spoof Detection: results from the first and second APOD redteam experiment taught us that arp spoofing is an important attack/tool for attackers to use in denying service or reading traffic on a network. We have developed an Arp Spoof detection mechanism that will notify an application of possible arp spoof activity on a network. The mechanism caches MAC/ip addess pairs and continually checks for changes in the pairs. If an attacker attempts to broadcast an invalid pair, then the mechanism will see this invalid pair and raise an alert. Currently, there is no qosket that is using this mechanism.

3.10.2 Requirements

1. Operating system: Unix (tested only on Linux)
2. arp and ping utilities
3. CORBA for Java (tested only with JacORB)

3.10.3 Detailed Explanation

Arp Spoof Detection mechanisms is implemented in the bus_containment qosket ($APOD_ROOT/qosket/bus_containment). The mechanism is implemented in two pieces; a java class, ArpSpoofDetectionSCImpl.java (a QuO SysCond) and a bash shell script, arpget.sh. The SysCond will call the shell script every specified number of seconds to get MAC/ip addresses. Those address pairs are compared against the previously received pairs for any changes. If there is change in any of the pairs the SysCond's value becomes true. This can be used in a contract as a sensor for possible malicious network behavior. The address pair that does not match the previous values will be available in the SysCond for any actuator mechanisms. The idl for the ArpSpoofDetectionSC is in $APOD_ROOT/qosket/bus_containment/idl.

ArpSpoofDetectionSC:

- Grabs the agility property from the properties
  Void init (java Properties file) –
- Set the amount of time between the shell script calls
  void setAgility(in long time)
- Set the name of the property list with subhost names
  void setSubnetHostListName(list name) -
- Returns the victim and attacker host ip addresses
  ArpSpoofPayloads getArpSpoofData() –
- Set an iptables system condition reference
  Void hookupActuator (reference to APOD's iptables SC) -

4 Prepackaged Reusable Defense Policies as Qoskets

A Qosket is defined independently of any application objects and represents a bundle of specifications and implementations of an adaptive behavior. In the case of APOD, a qosket contains a self-contained adaptive defense policy. Qoskets can be woven together with a functional specification and implementation to generate an adaptive delegate layer, or alternatively can be used standalone to integrate the behavior as part of the operating environment assisting the application in its defense. A defense qosket is each of the following, simultaneously:

- A software component, i.e. a defense qosket is a set of specifications and classes with an interface for implementing adaptive defenses. The defense qosket uses underlying APOD mechanisms.

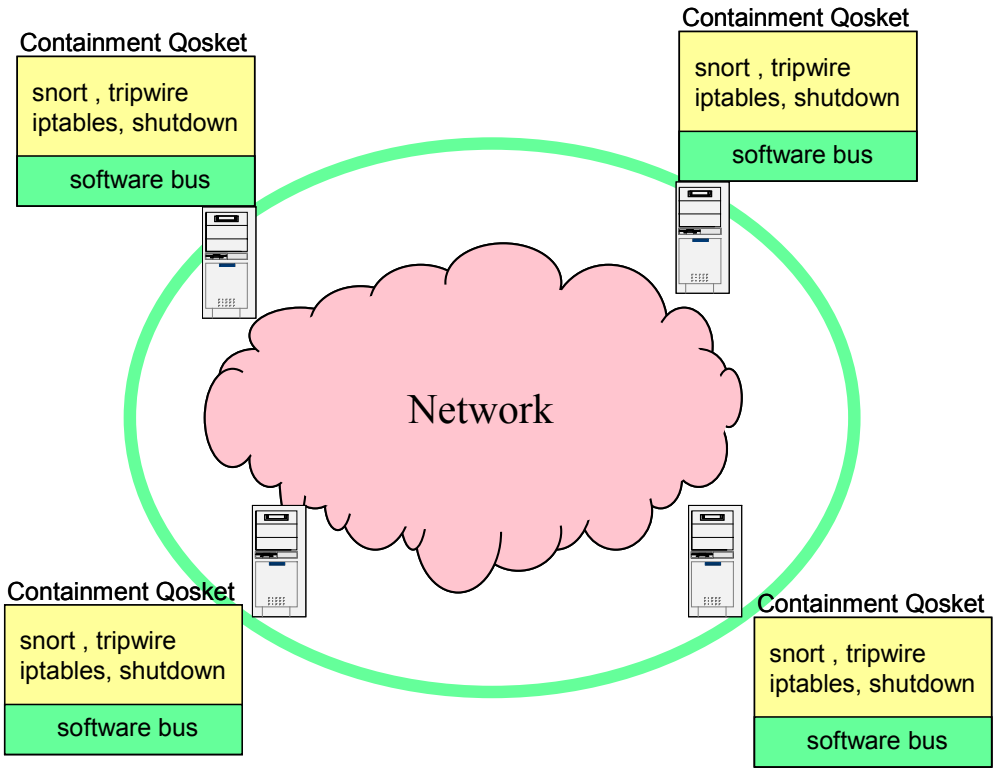
- A packaging of behavior - a defense qosket encapsulates elements of an adaptive defense behavior and a policy for using that behavior, in the form of contracts, measurements and code to provide adaptive behavior

- A unit of behavior reuse, largely focused on defense - i.e., a defense qosket can be used in multiple applications, or in multiple ways within a single application, but typically deals with defenses to security attacks.

The current defense qoskets in APOD are focused around the two aspects of attack containment and dynamic reconfiguration to evade attacks.

4.1 Containment Qosket - breaking the flow of attacks

The containment qosket implements a **coordinated quarantine** algorithm. It can be started out-of-band on application hosts, and tries to contain attacks once the have been detected. Since containment offers the possibility of self-inflicted DOS attacks, the containment qoskets run a distributed agreement algorithm using the software bus in an effort to prevent such cases.



We've also implemented a scaled down version of the containment qosket called appcontainment qosket. It does not have any local sensors, but rather just listens on the software bus for attacks detected by other qosket nodes and uses iptables to block traffic where appropriate.

4.1.1 Mechanisms Used

- Iptables to block all traffic from an attacker
- Snort to detect network based attacks
- Tripwire to detect corruption of file systems
- TCP Connection flood detection to respond to connection floods
- Execution manager to shut down hosts that have been infiltrated
- Software bus to run a coordination protocol between the qosket nodes and count the number of replica deaths

4.1.2 Adaptation Policy

The containment qosket implements the following high level adaptive defense policies:

- Detect network based attacks and respond by isolating the source of this traffic and consistently dropping all traffic from that source on all hosts. If the source of the attack is a host running the containment qosket, shut down the compromised host.
- Detect file system corruptions and respond by consistently dropping all traffic from the corrupted host on all hosts. In addition, shut down this compromised host.
- Count the number of replica deaths on each host. After exceeding a threshold, drop all traffic from this host on all hosts and shut it down.
- Note1: Shutting down of hosts only gets executed if we have enough hosts left.

Here a detailed description of the agreement algorithm used in this qosket.

Coordinated quarantine - This logic is implemented in the containment qosket. Since containment qoskets run only on broker factory hosts, when we say the containment qosket on H, H is always a broker factory host. If the containment qosket on a host H:

- Finds that H (i.e. itself) is marked SHUTDOWN, it shuts the host down. How a host is marked shutdown is described later.
- Finds that a host X is marked DROP, then it invokes the local iptables to block out traffic to and from X. How host is marked DROP is described later.
- Is told by Tripwire about a violation on H, it marks H with TWViolation and this marking is communicated to every containment qosket.
- Is told by Snort about a snort violation whose source is another broker factory host B, then it marks B with SNORTViolation and this marking is communicated to every containment qosket.
- Is told that 2 broker deaths were observed on H, it marks H with LDViolation, and this marking is communicated to every containment qosket.

Note: what follows is the crucial marking of shutdown and drop.

- Is told by Snort about a snort violation whose source is a non-APOD host X, it marks X as DROP, and that marking is communicated to every containment and appcontainment qosket.

- Finds that it is in a small partition (i.e. the only live hosts this qosket can see are those on its local lan, and one of them is a client or a server) then it marks H as SHUTDOWN.
- Is the leader, which is uniquely defined, and thinks that there are enough broker factory hosts available (as determined by a threshold), and there are suspected hosts (i.e. hosts marked SNORTViolation, TWViolation or LDViolation) then it goes into the following algorithm:
- Sort the dirty hosts in their numeric order (use int values of addresses).
- From this sorted list of dirty hosts, create a toKill list as follows:
    - The hosts that has all three violations are put first
    - The hosts that has either a TWViolation or a LDViolation are put next
    - Then the hosts that has SNORTViolation are put.
- Compute the number of hosts m that can be sacrificed (so that we only kill hosts up to a certain threshold) and mark the first m hosts on the toKill list with SHUTDOWN and DROP.
- If there are hosts in toKill that are not marked above, set HINTs for the selection qosket.
- If there are not enough broker factory hosts available and there are suspected hosts, then it sends HINTs for the selection qosket.

4.1.3 Customization Issues

In order to use the containment qosket in a custom application, the following files in APOD_ROOT/qosket/bus_containment have to be edited:

- contain.prp
- naming.prp

4.2 Dynamic Reconfiguration / Outrunning attacks

The bus_selecthost qosket implements a simple policy for outrunning an attacker by placing new replicas on different computers in different LANS (security domains). The default policy is that IF the last failure was observed on computer X in LAN Y, try to start new replicas on computer Z in LAN W, with Y != W, and Z is randomly selected among the available hosts.

We plan on enhancing the policy to do proactive dynamic reconfiguration, with the intend of delaying the attacker right from the start (instead of waiting for the first failure to occur).

4.2.1 Mechanisms Used

- Software bus to restart replicas

4.2.2 Adaptation Policy

Outrun

- If a broker dies, the selection qosket tries to compute a list of candidate hosts by selecting a broker factory host that is:
  - Not shut down (on shutdown list)
  - Not on the same subnet as the last crashed broker

  If no candidate is found, it does nothing.

  If there are candidates, then it tries to find a candidate that is not dirty (i.e. does have any HINT set). If it can find such a clean candidate, then that candidate is made the preferred host for starting brokers.

  If no clean candidate was found, a dirty candidate is made preferred host.
- If the selection qosket is sacrificed because of host quarantine, the bus will start a replacement.

4.2.3 Customization Issues

In order to use the selection qosket in a custom application, the following files in APOD_ROOT/qosket/bus_selecthost have to be edited:

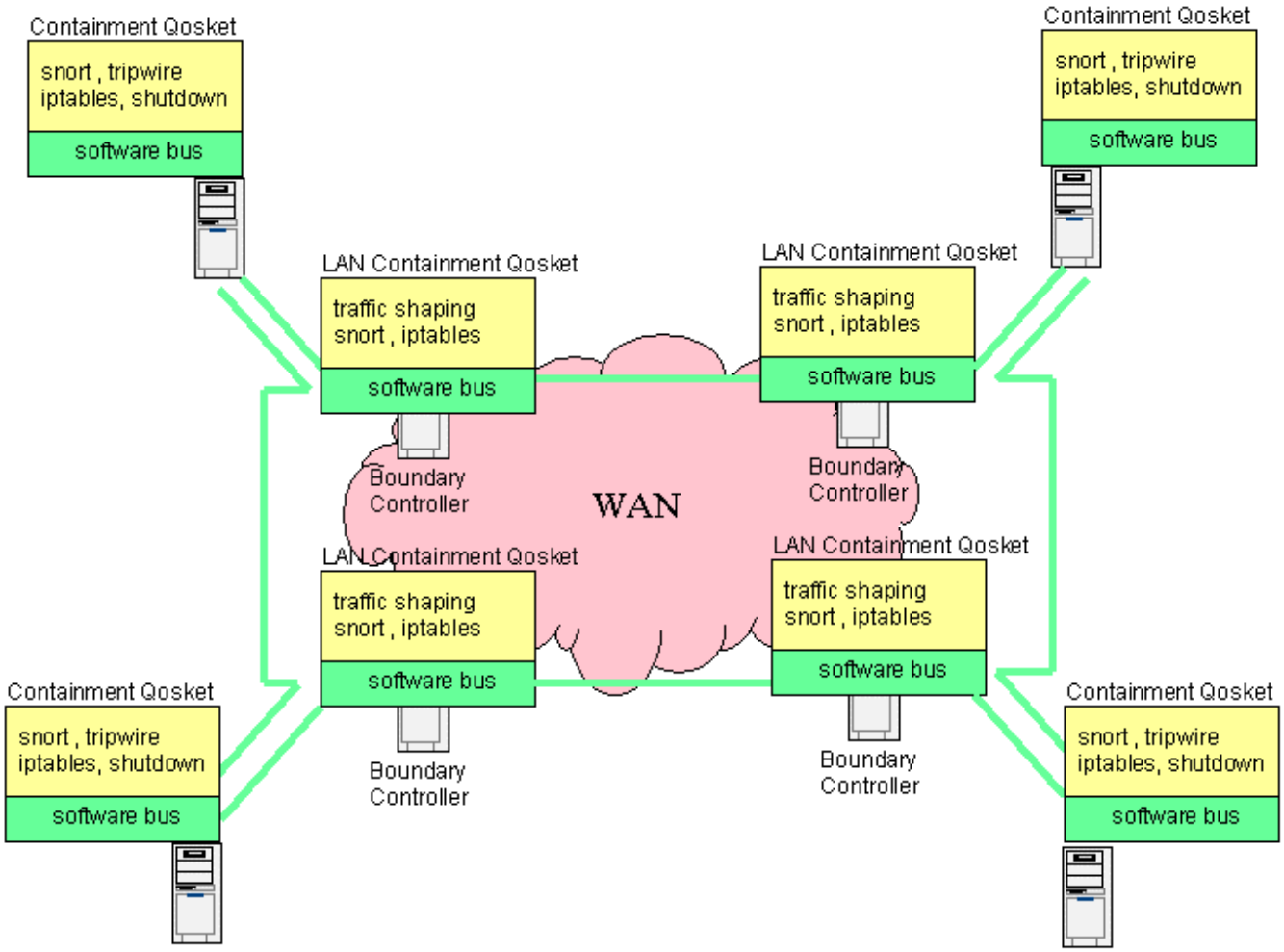
- selecthost.prp
- testdata.prp

4.3 LAN Containment Qosket - controlling floods and block closer to the source

The LAN Containment Qosket is a reusable middleware component for implementing a boundary controller system on Linux routers. This qosket participates in the coordinated quarantine policy of the containment qoskets and thereby implements an adaptive traceback mechanism for blocking attackers closer to their source. In addition, it gathers statistics about number of forwarded packets and number of forwarded bytes per second during a calibration phase, and then compares the actual throughput numbers to those values in an attempt to detect outgoing floods. If a flood is detected, the LAN containment qosket will rate limit outgoing traffic for a specific amount of time.

4.3.1 Mechanisms Used

- Iptables to block all traffic from an attacker
- Snort to detect network based attacks
- Software bus to run a coordination protocol between the qosket nodes



4.3.2 Adaptation Policy

If the qosket on boundary controller bc

- Detects a significant increase in outgoing traffic, it limits, for a constant period, the rate at which the router allows traffic to leave the LAN. This implements a flood containment policy.

- Learns from the bus that host H within its LAN has been marked suspicious by containment qoskets, it will block traffic from/to H. This implements a traceback containment policy to block hosts closest to their sources.

4.3.3 Customization Issues

In order to use the LAN containment qosket in a custom application, the following files in APOD_ROOT/qosket/LANcq have to be edited:

- lcq.prp

5 APOD Example Applications

The $APOD_ROOT/examples directory contains a set of examples that show how to use the APOD toolkit in defense-enabling: i.e. how to use the defense mechanisms in the $APOD_ROOT/mechanisms directory in the context of a defense strategy that is appropriate for a given application. The examples are meant to demonstrate the overall concept of defense-enabling, augmenting the textual documentation by source code that can be compiled and executed. The focus is not so much on demonstrating the example application itself, but rather on highlighting and demonstrating APOD specific features. By understanding the examples, you will learn how to:

- Devise a defense strategy for an application
- Use the capabilities of defense mechanisms in the context of your defense strategy
- Encode adaptive response and coordinate the defense mechanisms using the QuO adaptive middleware
- Compile and run defense-enabled applications

In course of doing so, we hope you will also learn how to:

- Develop adaptive distributed application in general
- Use QuO middleware and interface QuO with 3rd party products

5.1 brokered_imgsvc

This application implements a distributed image server application, containing of clients, brokers, and servers. Both clients and servers announce their presence to the brokers, which match up the two parties.

As part of an ongoing joint operation, a database of intelligence images must be made available to multiple analysts and planners in various locations who are planning a campaign of air assaults and follow up ground action. This operation is intended to neutralize an opponent who has nearly completed a suite of weapons of mass destruction; all indications are that the opponent will employ them against the United States and various allied countries as soon as the weapons become available. Hence, rapid completion of the planning process is essential.

In order to achieve this goal, a user must have low latency access to the image database. The opponent is known to have skilled information warfare specialists who are anticipated to take measures to degrade the planning process.

We have defense-enabled this application to stay survivable in the presence of such attacks. For that purpose, we've designed and implemented the following qoskets:

- Containment qosket to break the flow of attacks
- Dynamic reconfiguration qosket to outrun attacks
- LAN Containment qosket to control flooding and block closer to the source

See the chapter about Qoskets for more details on the adaptive policies.

See the chapter about red team experiments about experimental results from test runs with real-world attackers.

5.2 sniff_abatement

This example demonstrates defense-enabling in its simplest form. It also shows how defense-enabling can be done in increments. A defense-strategy often involves reactive adaptation where a sensor mechanism triggers an actuator mechanism. In this release we have left this example in an intermediate form where we trigger the actuator mechanism manually. In this stage the behavior of the application and the actuator can be studied without deploying the actual sensor mechanism and without having to launch real attacks that the actual sensor mechanism is supposed to detect.

5.2.1 Stepping through the defense enabling process

The application consists of a simple client and a server that communicate in the clear. The objective is to introduce a barrier at the very early stages of an attack. Although confidentiality is not initially a part of defense-enabling objectives, keeping things hidden from the attackers view makes it harder for the attacker to succeed. In order to attack an application, the attackers often sniff the network to find out information about the application such as on which host its components run, which ports they use etc. The idea here is to adapt the network transport transparently to the application when sniffing is suspected so that application specific traffic goes over a secured and encrypted channel.

Given this simple defense-enabling goal, we need two mechanisms to fulfill it:

1. A detection mechanism that indicates suspicious sniffing activity
2. A transport mechanism for carrying the application traffic in an encrypted form

Currently we use manual trigger. However, we have explored anti-sniffing/sniffer detection tools and in future we will use one such tool as the trigger. We use IPSec, specifically the FreeS/WAN(www.freeswan.org) implementation for Linux as the secured transport.

Refer to the application's README and source code for further details about how the defense-strategy of switching to IPSec is integrated with the application.

5.2.2 Description

When triggered, as part of the defense-strategy the communication between the client host and the server host is switched to IPSec. A manual trigger is also provided to switch back to IP from IPSec.

5.3 bette-3stage

This example demonstrates defense-enabling a simple video data service, where a display client requests for images from a server. The path between the display and the server can be attacked; also the server can be attacked. We consider display clients transients and disposable, therefore did not consider attacks on the client.

5.3.1 Stepping through the defense-enabling process

The purpose of the bette-3stage example is to demonstrate the combination of multiple defense mechanisms into a single application. In particular, this example defends against an attacker who

- Attempts to flood a network segment that is critical for the application and/or
- Attempts to disable the application by destroying key application components
- Given this goal, let us first identify the survivability requirements:
- Despite flooding, communication between key application components must continue
- Even if a key component is destroyed, other application components must be able to obtain the services it offered from somewhere else

This gives us a pretty good idea of what a useful defense-strategy for this application would look like:

- If flooding is suspected, establish a bandwidth reservation on the key segment
- If an object dies,
    - If there was a reservation associated with the dead object, cleanly teardown that reservation
    - Start another object
    - First we may restart on the same host, but if we observe repeated crashes on a host X, start on a different host assuming that the host X is being taken over by the attacker
    - If there was a reservation associated with the object that just died, re-establish the reservation with the replacement object that just started

Now we need to identify the defense mechanisms that provide the capabilities required to support such a strategy:

- The DIRM bandwidth management mechanism can establish reservation given two endpoints and a flow spec

- The bus mechanism can detect object crash and maintain a fixed number of replicas. It can also be commanded to avoid or prefer specific hosts for placing replicas

At this point all is left is to implement the defense-strategy and integrate it with the application. This is very similar to adding adaptive behavior to an application using QuO. In this particular case, we use the system conditions provided by the DIRM and BUS mechanisms along with other QuO system conditions: no custom system condition was required. However, a custom contract that provides the basis for adaptation and adaptation spec that captures the strategy was created using the QuO QDL. Refer to the README for this example and the source code in consultation with the appropriate QDL reference manual for further details.

QuO code generators are used to weave together this survivability aspect with the basic application. The README documents how to build this application from the source and how to run it. The behavior of the defense-enabled application is briefly described below.

5.3.2 Description

After starting the client-server application, the client starts requesting and displaying images from the server. The first attack is flooding a network link in the route between client and server. The application adapts by establishing a RSVP reservation. In the second attack, the attacker kills the server processes, while the flooding is continued. The application adapts by restarting a new server process on the same machine, and reestablishing the RSVP connection. After observing 2 server crashes, the application finally abandons the server host, and starts new server processes on a dedicated backup server.

5.4 warfare_scenario2

This example demonstrates the use of a relatively complicated defense-strategy in the context of a simple application that consists of a client and a server. One key thing to note in this particular case is how application level information can be effectively used to escalate the defense in a dynamic manner.

5.4.1 Stepping through the defense-enabling process

In this example, the threats we defend are:

1. Attack on the timeliness aspect of the application: if the client is forced to wait infinitely the application is effectively unusable (i.e. the attacker wins). The goal of the attacker is to attack the environment (network, OS, ORB) so that a call hangs without causing any exception or error.

2. Signature based attacks that can be launched from network.

The survivability requirements are:

1. The client should not wait infinitely even if no error or exception is observed.

2. If a threshold is reached it should try an alternative service provider that has better security

The survivability strategy we employed here is:

1. Use two service providers

2. Use a timeout on calls, when expires try the alternative server

3. When switching to a server S, activate or increase the sensitivity of IDS on the host that runs S

4. When IDS indicate suspicion, strengthen firewall on the host that runs S.

The defense mechanisms we used to implement the above strategy are:

1. An instance of application level assertion check (timeouts)

2. The Snort network based IDS

3. Iptables based packet filtering

For further details, refer to the README and source code of the warfare_scenario2 example.

5.4.2 Description

When the server's performance degradation (caused by the attacker) is caught by the application level assertion (timeout) mechanism, the client switches servers and notifies the Snort mechanism to monitor the server host.

If the Snort mechanism recognizes a known signature, it logs the packets, with the source address, parses the log files and determines if an attack is occurring. The iptables based packet filtering mechanism located on the attacked server is given the offending host name and port, with which it can block incoming packets from that host (and port).

The main purpose of this scenario is to show that we can recognize a degraded server and reactive to it by rerouting the client to another server and start an intrusion detection system to monitor the slow server.

6 Future Work

The APOD project concluded with a set of Red Team experiments to evaluate the defense enabling concept. These experiments used the brokered_imgsvc application (section 5.1), defense-enabled with a subset of the currently-available mechanisms, running in a laboratory environment. The experiments established that the APOD approach is able to increase the survivability of some distributed applications.

The APOD Toolkit described in this document is by no means complete. Potential future enhancements include:

1. Additional mechanisms: a couple of new defense-mechanisms are underway

2. Additional examples: examples showing more varieties of defense-enabling, using different defense-mechanisms

3. Packaging: packaging defense-mechanisms or defense strategies in easily (re)usable forms so that the full potential for middleware-based defense can be realized

6.1 Additional mechanisms

Some of the defense mechanisms that we have designed and experimented with earlier versions of Java and Visibroker include:

1. Security Enhanced Linux: "SELinux": this is a mechanism that aims to use the enhanced security capabilities of the SELinux operating system as an actuator.

2. Assertion checks: we have an instance of application specific assertion checking mechanism in the form of timeouts on client calls. The notion of timeout checking is described in the timer mechanism directory of the APOD software release. We are exploring other assertion checks that are good indicator of potential attacks.

6.2 Additional Examples

Potential candidates in this category are examples that we have released earlier with older version of third-party mechanisms used as defense-mechanisms as well as applications that use the new defense-mechanisms that are under development.

6.3 Packaging defense-strategies into Qoskets

Qoskets are packages of adaptive behavior (adaptation control and mechanisms whose services are needed for that adaptation) that are reusable. Defensive adaptation can be componentized into a qosket, making defense-enabling easier. In additions to the qoestets documented in Section 4, following behaviors are interesting candidates for being packaged into qoskets.

6.3.1 Change Server Object

The client initially communicates with Server1. Upon adapting, the QuO delegates routes a CORBA call to alternative remote object server 2. The client is therefore communicating with server 2. Aqua/bus can be used to synchronize the state of server 1 and 2.

Figure 8 Scheme of server switch-over adaptation qosket

6.3.2 Change of Network Routes

The client communicates with server 1 over network A (which includes the client's Ethernet card 1 and server's Ethernet card 1). Adaptation: Switch over communication between client and server to backup network B. For instance, network B includes the client's serial line interface 2 and the server's serial line interface 2.

Figure 9 Scheme of network changing adaptation qosket

7 APOD People and Acknowledgement

Michael Atighetchi (matighet@bbn.com)

Chris Jones (ccjones@bbn.com)

Partha Pal (ppal@bbn.com)

Paul Rubel (prubel@bbn.com)

Franklin Webber (franklin@eutaxy.net) (Principal Investigator)

APOD team would like to thank Dr. Douglas Maughan (DARPA ATO) for supporting this project, Dr. Richard Schantz and Dr. Joseph Loyall (BBN) for technical guidance and advise.